

# C++ Language Reference for CMIS 102

## Contents:

1. Introduction .....	2
2. Comments.....	2
3. Constants .....	3
4. Identifiers.....	3
5. Reserved Words.....	4
6. Primitive Data Types.....	4
7. Variables and Declarations.....	5
8. Operators .....	5
9. Expressions.....	5
10. Assignment Statements .....	7
11. If...Else Statements .....	7
12. Switch Statements .....	8
13. While Loops .....	9
14. For Loops.....	9
15. Input and Output Statements .....	10
16. Compound Statements.....	11
17. Arrays .....	11
18. Function Declaration and Invocation .....	12
19. Return Statement .....	12
20. Class Method Invocation.....	13
21. Compiler Directives .....	13
22. Program Structure.....	14
Appendix A. Syntax Summary .....	15
Appendix B. Common Mistakes .....	18

# The C++ Programming Language: A Subset for CMIS 102

## 1. Introduction

The goal of CMIS 102 is to teach you some basic concepts about programming that you can take onward to more advanced courses, regardless of the programming language used in the more advanced courses. Of course, it is impossible seriously to learn about programming without learning a programming language at the same time. Conversely, it's not possible to learn your first programming language without learning how to program at the same time. So we have had to choose a language. We have chosen C++. C++ is a major language in its own right, and in syntax and control structures is very similar to Java.

The C++ programming language, defined in ANSI/ISO Standard 14882-2003, is very large and complex. But for CMIS 102 use, within C++ we can find a small subset that is both easy to learn, and has upwards of 80% commonality with Java. This note is an informal but reasonably precise description of this subset.

Bear in mind that C++ compilers recognize the full C++ language. So it is possible for you to write code that goes beyond what we describe here, and still be accepted by the compiler. But if you write code that adheres to the following rules, your program should compile correctly. Whether it does what you want it to do is, of course, another question.

## 2. Comments

You can include comments in your source code in either of two ways:

- On any line of code, anything after `//` is a comment.
- Anything between `/*` and the next `*/` is a comment. This format allows multi-line comments.

Examples:

```
x = x + 1; // Statement with comment
/* Here is the first line of a multi-line comment
   and here is more of it
   and here is the rest of it.
*/
```

### 3. Constants

An *integer constant* is a string of one or more digits, optionally preceded by a minus sign.

Examples: 137, -456, 0

A *float constant* consists of an optional minus sign, a string of one or more digits, a decimal point, and a string of one or more digits.

Examples: 123.456, -12.34, 0.0

A *boolean constant* is either `true` or `false`.

Examples: `true`, `false`

A *character constant* is either (a) a character other than `\`, `"`, or `'`, enclosed in single quotes, or (b) one of `'\n'`, `'\t'`, `'\\'`, `'\"'`, or `'\''`, indicating newline, tab, backslash, double quote, and single quote, respectively. These forms of character constants are called *escape sequences*.

Examples: `'A'`, `'4'`, `'b'`, `'\n'`, `'\t'`, `'\\'`, `'\"'`, `'\''`,

A *string constant* is a string of characters enclosed in double quotes. The characters may not include `\`, `"`, or `'`; but any of the escape sequences listed above may be included.

### 4. Identifiers

An *identifier* is a name given to a variable or function. An identifier can contain only letters, digits, and underscore, and cannot begin with a digit.

Examples: `Cat`, `Dog789`, `EXIT_SUCCESS`, `_888Hello`

Caution: Identifiers are case sensitive. `Cat`, `CAT`, and `cat` are all different.

Tip: Although it is permitted, you should never begin an identifier with an underscore, or use two underscores in a row. Doing either may create a conflict with a library function.

## 5. Reserved Words

Certain identifiers have specified meanings in the language. You may not redefine them. Table 1 lists the 73 reserved words of C++. You will recognize many of them (those in yellow). The others are listed here just so that you will know not to use them as identifiers in your programs.

**Table 1. C++ Reserved Words**

and	and_eq	asm	auto
bitand	bitor	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	not	not_eq	operator
or	or_eq	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	xor
xor_eq			

## 6. Primitive Data Types

A *data type* is the set of values that a variable of that type can hold.

A variable of type `int` can hold any integer in the range  $-2,147,483,648$  to  $+2,147,483,647$ .

Note: This statement assumes a word length of 32 bits, which is typical.

A variable of type `float` can hold any one of about 4,294,967,296 specific rational numbers. The largest positive value of a `float` variable is about  $3.388 \times 10^{38}$ . The smallest positive value of a `float` variable is about  $1.413 \times 10^{-45}$ . A variable of type `double` can hold any one of about  $1.84 \times 10^{19}$  specific rational numbers. The range of positive `double` values is about  $4.9 \times 10^{-324}$  to  $1.7 \times 10^{308}$ .

Note: This statement assumes a word length of 32 bits, which is typical.

Disclaimer: The exact number of allowed values actually depends on the technique for representing numbers used in the particular computer. For a computer using the IEEE 754 standard, the number of different `float` values is somewhat less than 4,294,967,296, for technical reasons.

A variable of type `bool` can hold either `true` or `false`. Theoretically, it only takes one bit to hold a value of type `bool`, although most compilers allocate more, such as 8, 16, or 32.

A variable of type `char` can hold an integer in the range 0 to 255. The values in the range 32 to 126 represent printable characters (the so-called ASCII character set.)

## 7. Variables and Declarations

A *variable* is a name assigned to a portion of your computer's memory that can hold a value of a specific data type. A declaration is a program statement that asks the compiler to assign a portion of the computer's memory to hold a data item, and to associate with it the name you specify. You must declare every variable you use.

Examples 1: `int j, numberOfPeople;` requests the allocation of an appropriate amount of memory (usually 32 bits) to hold each of two values of type `int`. These memory locations will be known by the names `j` and `numberOfPeople`.

Example 2: `float myWaterConsumption;` requests the allocation of an appropriate amount of memory (usually 32 bits) to hold a value of type `float`. This memory location will be known by the name `myWaterConsumption`.

Note: Although it is an engineering issue rather than a programming issue, whenever you deal with values of data type `float`, you need to specify the units. For example, `myWaterConsumption` may be in cubic feet per hour, or liters per day, or whatever.

## 8. Operators

Operators are actions that the computer can perform on one or two inputs called *operands*. Each operand must be of some data type. The data type of the result may depend on the data type of the operands. Table 2 lists the C++ operators, their symbols, their input data types, and their result data types. They are listed in equal-precedence groups, in order of decreasing precedence.

## 9. Expressions

An expression is a combination of variables, constants, operators, and parentheses, such that if values are assigned to each of the variables, the expression can be evaluated to produce a value of some data type. The precise rules for constructing valid expressions are rather complicated, but they are close enough to the rules for expressions in algebra that you should be able to understand the concept from some examples.

The following are all valid expressions:

```
x + 5
(x + 5) - y
(x + 5) / (2 * y + 3)
(x + 37 <= 40) || (y != 18)
```

**Table 2. C++ Operators**

<b>Operator Name</b>	<b>Symbol</b>	<b>Unary or Binary</b>	<b>Input Data Type</b>	<b>Result Data Type</b>
Increment	++	U	int	int
Decrement	--	U	int	int
Unary minus	-	U	int	int
Unary plus	+	U	float	float
			int	int
NOT	!	U	bool	bool
Multiplication	*	B	int	int
			float	float
Division	/	B	int	int
			float	float
Remainder	%	B	int	int
Addition	+	B	int	int
Subtraction	-	B	float	float
			int	int
Less than	<	B	int	bool
			float	bool
Less than or equal	<=	B	int	bool
			float	bool
Greater than	>	B	int	bool
			float	bool
Greater than or equal	>=	B	int	bool
			float	bool
Equals	==	B	int	bool
			float	bool
Not equals	!=	B	bool	bool
			int	bool
			float	bool
			bool	bool
AND	&&	B	bool	bool
OR		B	bool	bool

## 10. Assignment Statements

An assignment statement has the form *variable = expression;*. Execution of an assignment statement consists of evaluating the expression, and then storing the resulting value in the variable. The result of evaluating the expression must be a data type compatible with the variable.

Example 1: Assignments to a some variables of type `float`

```
float x, y, z;  
x = 3.0; y = 5.6;  
z = (x + y) / (x - y);
```

Example 2: An assignment to a variable of type `bool`

```
bool answer; int x, y;  
answer = x < y;
```

It is possible to combine a declaration statement with an assignment statement. This is called *initialization*.

Examples:

```
float x = 3.4, y = 5.6;  
is equivalent to  
float x, y;  
x = 3.4; y = 5.6;
```

## 11. If...Else Statements

An "if...else" statement is of the form `if (logicalExpression) statement1 else statement2` or, if `statement2` is "do nothing", simply of the form `if (logicalExpression) statement1`. *logicalExpression* must be an expression that evaluates to a value of type `bool`. It operates as shown in the flowchart to the right.

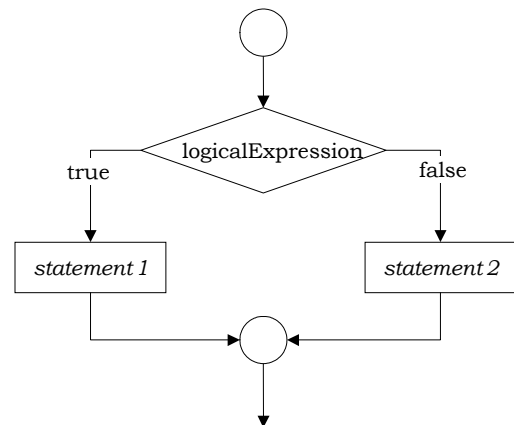
Examples:

```
if (x < y) z = +1; else z = -1;
```

```
if (x >= y && y >= z) {  
    a = 5; b = 6;  
}
```

Note: *statement 1* and *statement 2* are often compound statements. *statement 2* can be empty, in which case the word `else` is omitted. Often, *statement 2* is itself an if...else statement, as in the following example

```
if (g >= 70) grade = 'A'; else if (g >= 80) grade = 'B';
```



## 12. Switch Statements

A switch statement is of the form

```
switch (expression) {
  case value1:
    sequence of one or more statements
    break;
  case value2:
    sequence of one or more statements
    break;
  ...
  case valueN:
    sequence of one or more statements
    break;
  default:
    sequence of one or more statements
    break;
}
```

The following rules apply to switch statements:

- *expression* must evaluate to an integer.
- Each of *value1* through *valueN* must be an integer constant.
- The default clause is optional.

When the switch statement is executed, *expression* is evaluated and compared to *value1* through *valueN*. If a match is found, the statements following the match are executed up to the `break;` statement. If there is no match, the statements following *default:* are executed.

Example:

```
switch (j) {
  case 1:
    z = x;
    break;
  case 2:
    z = x*x;
    break;
  default:
    z = 1;
    break;
}
```

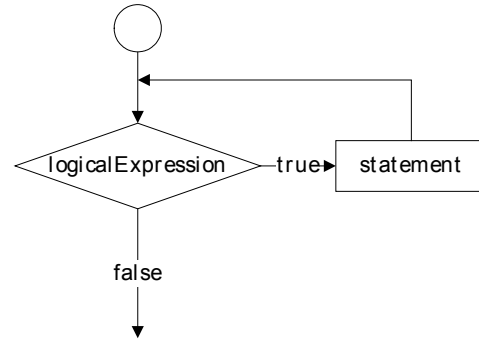
Caution: If you omit a `break;`, execution of statements will continue into the next group, which is almost certainly not what you intend.

### 13. While Loops

There are two types of "while" loops: pre-test and post-test. In the following, *logicalExpression* must be an expression that evaluates to a value of type `bool`.

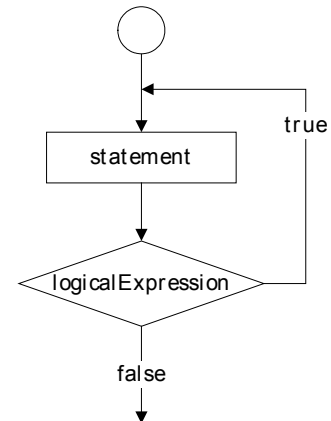
The pre-test "while" has the form `while (logicalExpression) statement`. It operates as shown in this flow chart:

Example:  
`while (n < 137) n = n + 2;`



The post-test "while" has the form `do statement while (logicalExpression);` It operates as shown in this flow chart:

Example:  
`do n = n + 2; while (n < 137);`

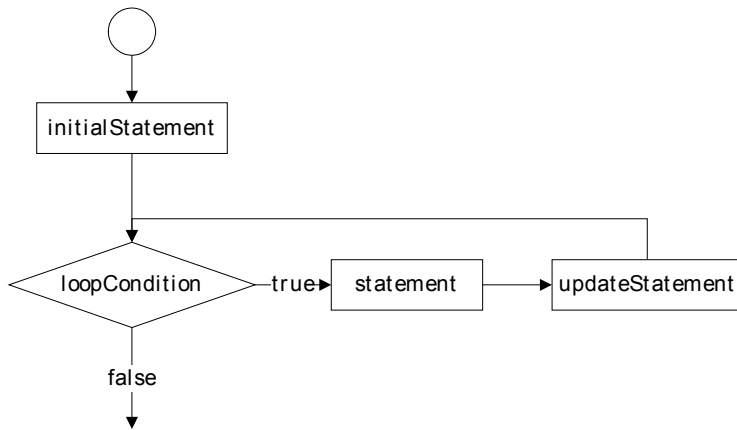


### 14. For Loops

A "for" loop has the form `for (initialStatement; loopCondition; updateStatement) statement`.

#### Explanations:

- (Step 1) The "for" loop begins with execution of the *initialStatement*.
- (Step 2) If the *loopCondition* is `true`, then *statement* (the body of the loop) is executed.
- (Step 3) The *updateStatement* is executed
- (Step 4) The cycle is repeated from (Step 2) until the *loopCondition* is `false`.



Example 1:

```
for (i = 0; i < 10; i++) x = x + 1;
```

In this example, the statement `x = x + 1;` is executed ten times, with the variable `i` ranging from 0 to 9.

Example 2:

```
for (j = 3; j <= 10; j = j + 2) x = x + 1;
```

In this example, the statement `x = x + 1;` is executed four times, with the variable `i` taking on the values 3, 5, 7, and 9.

## 15. Input and Output Statements

The statement to write output to the console has the form

```
cout << item1 << item2 << ... << itemN;
```

where each of `item1`, ..., `itemN` is either

- an integer, float, character, or string constant, or
- an expression that evaluates to an integer or float value, or
- `endl` (indicating newline).

Examples:

```
cout << "Hello";
cout << "The answer is " << x - y << " cubic yards." << endl;
```

The statement to obtain one or more `int` or `float` values from the user has the form

```
cin >> variable1 >> variable2 >> ... >> variableN;
```

where each of `variable1` through `variableN` is of type `int` or `float`.

Examples:

```
int i; float x;
cout << "\nPlease enter a value for i: ";
```

```
cin >> i;
cout << "\nPlease enter values for x and y: ";
cin >> x >> y;
```

Warning: `cin` does no data validation. If you enter data that is not a valid integer or float, as applicable, a nonsense value may be assigned to *variable*; or, the computer may go into an infinite loop, forcing you to shut down the program.

## 16. Compound Statements

A *compound statement* is a sequence of one or more statements enclosed in braces. The format of a compound statement is `{ statement1 statement2 ... statementN }`.

Example: `{x = 1; y = 2; if (x < y) {z = 1; w = 2;}}`

Note: This example shows a compound statement inside an "if" statement which is in turn inside a compound statement. This is called *nesting*.

## 17. Arrays

An *array* is a sequence of values of the same type, stored in consecutive locations in the computer's memory. If the number of elements in the array is  $N$ , then the values are numbered  $0, 1, 2, \dots, N-1$ .

The form of an array declaration is

*element-type array-name[array-size];*

where

*element-type* is a valid type, such as `int`, `float`, or `char`;

*array-name* is an identifier

*array-size* is a positive integer constant

Example: `float myArray[20];`

An array element reference has the form

*array-name[subscriptExpression]*

where

*array-name* is the name of the array, and

*subscriptExpression* is an expression that evaluates to an integer in the range  $0$  through  $N-1$  (where  $N$  is the size of the array).

An array element reference can be used in an expression, or it can be the target of an assignment.

Example:

```
myArray[0] = 2*myArray[1] + 3*myArray[2];
```

## 18. Function Declaration and Invocation

A function declaration is of the form

```
return-type function-name(parameter-type-1 paramater-name-1,  
                           ..., parameter-type-N paramater-name-N)  
{  
    your statements  
}
```

The rules are:

- *return-type* is the type of value to be returned: either `int`, `float`, or `char`; or `void` if no value is to be returned.
- *function-name* is an identifier to be used as the name of the function.
- *parameter-type-1* through *parameter-type-N* are the types of the formal parameters: `int`, `float`, or `char`.
- *parameter-name-1* through *parameter-name-N* are identifiers by which the parameter values will be referred to within this function.
- If the return type is other than `void`, there must be a `return` statement in the body of the function.

A function you have defined is invoked by writing

```
function-name (actual-parameter-1, ..., actual-parameter-N)
```

The rules are:

- *actual-parameter-1* through *actual-parameter-N* are expressions whose types are *parameter-type-1* through *parameter-type-N* respectively.
- You can use a function invocation as a statement, by putting a semicolon after it; or
- If the function returns a value, you can use it in an expression just as you would use a variable.
- If the function returns a value, you must have a `return` statement in the body of the function.

Example:

```
int biggerPlusOne(int x, int y) {  
    if (x > y) return x + 1; else return y + 1;  
}  
  
. . .  
int x, y, z;  
. . .  
z = 2*biggerPlusOne(x, y) + 5;
```

## 19. Return Statement

The format of a return statement is `return value`; where *value* is an expression that evaluates to a value of the return-type of the function. See the example above.

## 20. Class Method Invocation

A *class* is a programmer-defined data type. A class is usually provided with a set of operations, called *methods*, that can be performed on values, or *objects*, having that data type. Invoking a method is similar to invoking a function, except that the name of the method is preceded by the variable name and a period.

Example:

Assume that the class `MyClass` has a method `getHeight` that returns a `float`. Then we can write

```
float x;  
MyClass xyz;  
x = xyz.getHeight();
```

## 21. Compiler Directives

Compiler directives provide information to the compiler that the compiler needs to process your source code. Compiler directives are NOT statements in the C++ programming language, so C++ syntax rules do not apply to them.

Caution: compiler directives do NOT end in semicolons.

`#include` directives tell the compiler to read C++ statements from some file. There are two forms, as shown in the examples.

Example 1: `#include <iostream>` tells the compiler to read information from the `iostream` file, which is to be found in a folder of similar files provided with the C++ compiler system.

Example 2: `#include "SStream.h"` tells the compiler to read information from the `SStream.h` file, which is in the user's own folder.

`#define` directives tell the compiler to replace every occurrence of an identifier in the program with a specific constant, as though the programmer had written the constant herself.

Example 1: `#define EXIT_SUCCESS 0`

Example 2: `#define PI 3.1415926535`

## 22. Program Structure

The structure of a simple C++ program is illustrated by the following example. Some remarks are included as comments.

```
#include <cstdlib>
#include <iostream>
// Here insert any other #include statements needed.
// Also insert here any #define statements you would like.

using namespace std;

// Here insert any functions that you have written
yourself.

int main(int argc, char *argv[])
{
    // Your declarations and executable statements here...

    system("PAUSE"); // Needed only for Dev-C++ compiler
    return EXIT_SUCCESS;
}
```

## Appendix A. C++ Syntax Summary

Language Feature	Form	Example(s)
Rest-of-line comment	// ...	// This is a comment
Multi-line comment	/*...*/	/* This is a comment that continues here.*/
Integer constant		0, 3, -5
Float constant		1.23, -4.56
Boolean constant		true, false
Character constant	'_' and escape codes	'A', '\$', '\n'
Identifier	Contains letters, digits, and underscore only; cannot begin with a digit	Cat, myVariable_5
Assignment	<i>variable</i> = <i>expression</i> ;	x = y + (2*z - 6/w) ;
If	if ( <i>logicalExpression</i> ) <i>statement1</i>	if (x < y) z = 5;
If...else	if ( <i>logicalExpression</i> ) <i>statement1</i> else <i>statement2</i>	if (x < y) x = 5; else z = 6;
Switch	switch( <i>expression</i> ) { case <i>value1</i> : <i>statements</i> break; ... case <i>valueN</i> : <i>statements</i> break; default: <i>statements</i> break; }	switch (j) { case 1: z = x; break; case 2: z = x*x; break; default: z = 1; break; }
While loop - pretest	while ( <i>logicalExpression</i> ) <i>statement</i> .	while (n < 137) n = n + 2;
While loop - posttest	do <i>statement</i> while ( <i>logicalExpression</i> ) ;	do n = n + 2; while (n < 137);
For loop	for ( <i>initialStatement</i> ; <i>loopCondition</i> ; <i>updateStatement</i> ) <i>statement</i>	for (i = 0; i < 10; i++) x = x + 1;
Output to display	cout << <i>item1</i> << <i>item2</i> << ... << <i>itemN</i> ;	cout << "Volume is " << x << " cubic yards." << endl;

Language Feature	Form	Example(s)
Input from keyboard	<code>cin &gt;&gt; variable1 &gt;&gt; variable2 &gt;&gt; ... &gt;&gt; variableN;</code>	<code>cin &gt;&gt; x &gt;&gt; y;</code>
Compound statement	<code>{ statement1 statement2 ... statementN }</code>	<code>{x = 1; y = 2; if (x &lt; y) {z = 1; w = 2;}}</code>
Array declaration	<code>element-type array- name[array-size];</code>	<code>float myArray[20];</code>
Array element reference	In an expression, or as target of an assignment	<code>myArray[0] = 2*myArray[1] + 3*myArray[2];</code>
Function declaration	<code>return-type function-name(parameter- type-1 parameter-name-1, ..., parameter-type-N parameter- name-N) {     statements }</code>	<code>int biggerPlusOne(int x, int y) {     if (x &gt; y)         return x + 1;     else return y + 1; }</code>
Function invocation	<code>function-name (actual- parameter-1, ..., actual- parameter-N)</code>	<code>... biggerPlusOne (v, w)...</code>
Return	<code>return value;</code>	<code>return x + 5;</code>
Include library header	<code>#include &lt;library-name&gt;</code>	<code>#include &lt;iostream&gt;</code>
Include custom header	<code>#include "file-name"</code>	<code>#include "xyz.h"</code>

Language Feature	Form	Example(s)
Program structure	<pre>#include &lt;cstdlib&gt; #include &lt;iostream&gt; // Insert other #include statements // Insert #define statements  using namespace std;  // Insert any function definitions  int main(int argc, char *argv[]) {     // Your declarations and executable     statements      system("PAUSE"); // Needed only for     Dev-C++     return EXIT_SUCCESS; }</pre>	

### C++ Operators

The operators in each group have equal precedence. Groups are listed in order of decreasing precedence.

Operator Name	Symbol
Increment	++
Decrement	--
Unary minus	-
Unary plus	+
NOT (Boolean op.)	!
Multiplication	*
Division	/
Remainder	%
Addition	+
Subtraction	-

Operator Name	Symbol
Less than	<
Less than or equal	<=
Greater than	>
Greater than or equal	>=
Equals	==
Not equals	!=
AND	&&
OR	

## Appendix B. Common Mistakes

*Listed below are the most common mistakes beginners make when writing C++. If when you compile your program you get lots of error messages, or error messages you can't figure out, read your program carefully to see if you have made any of these.*

**Writing C++ keywords with an initial capital**, such as `If` for `if`, `Else` for `else`, or `While` for `while`. Using any of `If`, `Else`, `While`, and others will probably result in "undefined symbol" messages.

**Writing any of several words, commonly used in pseudocode, that are not used in C++**, such as `set`, `declare`, `then`, and `endif`. Using these will probably result in "undefined symbol" messages.

**Forgetting the semicolon at the end of a statement that does not end in `}`**: Remember that every statement in C++ ends in either `;` or `}`.

**Declaring a variable with one capitalization pattern, and then using it with another**, such as: `int myHeightInInches; myHeightinInches = 63;`. This will result in an "undefined symbol" message.

**Forgetting the closing double-quote (`"`) at the end of a string**. Make sure that your double-quotes come in pairs. Forgetting a matching double-quote can result in any of a variety of error messages, many of which may be misleading, as the compiler is still looking for the missing double-quote!

**Forgetting the single-quotes around a character constant**, such as writing `W` when you mean `'W'`: This usually results in an "undefined symbol" message.

**Writing a character constant with double-quotes, or a string constant with single-quotes**.

**Using "smart" single-quotes around a character constant, instead of "plain vanilla" single quotes**: If your character constants look like `'A'` or ``A`` instead of `'A'`, the compiler will not recognize them as a character constants. This problem usually results from using a word processing program (such as Microsoft Word) to write your program, instead of Notepad or the development environment's own text editor. (A similar problem would result from using smart double-quotes around a string constant.)

**Using the assignment operator `=` where the equality comparison operator `==` is intended**: For example, writing `if (x = y) doSomething();` where `if (x == y) doSomething();` was intended. This error is not detected by the compiler, and can result in truly bizarre behavior of the program.

**The inadvertent Do-Nothing statement:** A semicolon standing alone is a valid statement, and it does nothing. This will normally not cause a compile time error message, but may result in incorrect operation. Consider the example,

```
int x = 5; while (x != 0);
```

The format of a while statement is `while (condition) statement`. In our example, *statement* is `;` and does nothing. So `x` remains equal to 5 and the while loop loops forever.

**Not enough parentheses when using logical operators and comparison operators:** A good example of this is writing `if (!x < y) {whatever}`. If you write this, you almost certainly mean `if (!(x < y)) {whatever}`. However, C++ holds that `!` has higher precedence than `<`, and interprets what you wrote as if you intended `if ((!x) < y) {whatever}`. Nonsense, you say! But C (and by extension C++) is a language written by programmers for programmers, and the compiler assumes that you meant exactly what you wrote. So (assuming `x` and `y` are `ints`), the compiler generates code to

- converts the integer `x` to Boolean using the rule that zero represents false and any nonzero number represents true
- negates the Boolean result, producing true or false respectively, and represents these by 1 and 0 respectively
- converts these to integers in the obvious way, and
- compares the resulting integer to `y`.

which is not at all what you intended.

(This unfortunate behavior was present in the predecessor language C, and was retained in C++ for backward compatibility. It does not occur in Java. In Java, the code `int x = 5, y = 6; boolean b = !x < y;` results in the error message "cannot be applied to int.")